CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Regression Testing Selection and Dependency Analysis: A Design Science Study

Bachelor of Science Thesis in Software Engineering and Management

Pontus Laestadius
Isabelle Törnqvist

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

{Regression Testing Selection and Dependency Analysis}
[A design science study]

Supervisor: Richard Torkar
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Regression Testing Selection and Dependency Analysis: A Design Science Study

1st Pontus Laestadius
dept. of Computer Science and Engineering
Gothenburg University
Gothenburg, Sweden
pontus.laestadius@gmail.com

2nd Isabelle Törnqvist
dept. of Computer Science and Engineering
Gothenburg University
Gothenburg, Sweden
gustornqis@student.gu.se

***Abstract*—Continuous Large Scale Software Systems (LS3) development depends on Regression Testing Selection (RTS) to uphold quality of the software in a cost-effective manner. Prioritization-, minimization- and selection techniques are examples of this. We explore an existing technique and then classify it as a selection technique. We design and develop a new artefact, with the purpose of comparing the two Regression Testing Selection Techniques (RTST) with a select number of metrics. We compare the RTSTs using 500 commits with a one-tailed t-test which shows non-significant results. The artefact shows lower standard deviation for all metrics compared to the existing technique. The artefact identified, and removed 151 false positives that the existing technique would perform dependency analysis on. We conclude that the artefact shows improvements over the existing technique, while the random technique performed the worst.**

***Index Terms*—software testing**

I. Introduction

Software testing is significant in assuring software quality [1]. Regression Testing (RT) is a form of qualitative software testing that gives confidence and validates modified software, and compares it with the previously established code base [2].

Rerunning all tests provides test coverage which is all encompassing and has high reliability [2]. On the other hand, executing all test cases is often expensive [3], as the cost scales with the size of the test suite [4].

Regression Testing Selection (RTS) finds a subset of tests from the entire test suite, when testing modified software [5]. RTS often comes at the expense of time, resources and test coverage [5]. The goal of RTS is that it should be cost-effective to perform, meaning the resulting costs must be lower than rerunning the entire test suite [5].

Hypotheses and implementations regarding RTS have coined the term Regression Testing Selection Techniques (RTST) [2], [6]. Prior research to examine and find the most efficient RTST has been done [1], [2], [5]. RTSTs focus on different qualities, often test coverage is the trade-off, as RTST are either *safe* or not [2]. Graves *et al*. defines a Safe RTST as a technique that guarantee that the subset will always reveal introduced software faults [2].

For continuous Large Scale Software Systems (LS3) development, developers frequently introduce new code to the main code-base [7]. Frequently introducing changes requires Continuous Integration (CI) to uphold quality [8]. CI performs activities to detect software faults (including RT), and becomes an essential part of maintaining the system. LS3 often contains larger test suites than smaller software systems. In order to reliably and continuously modify the LS3, code changes must be thoroughly tested. As the LS3 grows, fewer tests of the whole suite are executed. Segmenting the system into small components to isolate the test suites reduces the testing scope for the overall suite, and lower the cost of RT [9].

This study is conducted with Ericsson, one of the world's largest telecommunication companies. Ericsson develop LS3.

With this study, we aim to extend the current knowledge of RTST for LS3. This is done by comparing two implementations of two different RTST, as well as compare to an implementation of a random technique. For researchers, this study may provide additional data on applying RTSTs on LS3.

We pose the following research questions:

RQ 1: What are the RTST's performance and limitations?
RQ 2: How do the results differ when comparing different RTST on LS3 development?

Following the Design Science Research Methodology (DSRM) [10], we conduct an exploratory investigation into a RTS for a LS3. We use the findings from the investigation to create a new artefact, which aims to improve the existing implementation. The two implementations are compared on the LS3. Lastly, we apply statistical analysis to conclude the improvement.

We present the artefact to the stakeholder in an evaluation interview, showing non-significant results between the existing solution and the produced artefact, but we point towards a reduced total dependencies in favor of the produced artefact.

For practitioners, this study will show the potential benefits of implementing a RTST for dependency analysis.

This paper is structured as follows: Section II outlines related background literature. Section III describes the research methodology. Section IV presents the results and the artefact. Section V discusses all findings. Section VI summarizes the study.

II. Background/Related Work

For this specific product, we chose to focus on a single programming language, C/C++. Previous research show that file types are important when attempting to detect faults in code, and the authors identify that .hpp files produced 80% of test failures for their context [8].

This study has similarities to the RTS research field. However, we will not be doing an analysis on the test suites or test cases, but rather apply the RTS background to dependency analysis. Because of that, it is important to note that test suites are excluded by default.

To limit the scope of this study, we only account for Software Component (SC)s that specify their build dependencies, and not larger modules which encapsulate many SCs.

A. Regression Testing Selection Techniques

Technique-specific trade-offs are explored in a survey on 159 papers [4], where classes of techniques from the area of regression test optimization are presented, as well as a detailed analysis on them. Following are summaries of common techniques.

The definition of a safe technique varies slightly. Skoglund and Runeson propose a safe technique called firewall, which selects only modified and affected units of a System Under Test (SUT) that needs to be re-tested [11]. Orso *et al.* defines safe as a technique which will select tests that shows inconsistencies in the SUT [5]. Graves *et al.* however, defines safe as a technique that has complete fault detection [2].

In the empirical study by Graves *et al.*, it was found that in some cases the safe technique could not reduce the number of test cases and selected all [2]. In this study, we perceive a safe technique as one that will find any potential faults introduced by the commit.

Graves *et al.*'s study provide examples on implementation of the different RTSTs. The experiment also provided some insight into potential results of using the different RTSTs. However, due to their artefact representativeness not reflecting common systems found in industry, the results are not generalizable to other systems [2].

Minimization techniques aim to remove redundant test cases from the suite. In an empirical study on RTSTs, minimization was described as a test selection method that covers every segment that is reachable by the modified segment [2].

Contradictory conclusions were found with previously published results, regarding the effects on fault-detection capabilities when reducing a test suite. According to the study by Yoo and Harman, some of the studies they examined showed that removal of test cases had no drawbacks on fault-detection effectiveness. However, on SUTs where test cases had low block coverage, some studies pointed to that a reduced test suite will provide less effective fault-detection [4].

Yoo and Harman suggests that these contradictory results can be due to different contexts of the SUT. These context factors are, for example, size and structure of the SUT. It is also suggested that these contradictory results are evidence that its difficult to generalize an answer on test suite minimization for any SUT. In Graves *et al.*'s study, it was found that minimization was the least effective. Random selection had the same fault detection coverage, with less cost, although only half of the time.

According to Yoo and Harman, a selection technique adds tests to the suite, based on modifications to the software [4]. This description matches a technique that Graves *et al.* refers to as a dataflow technique. We found that different literature uses different terms to refers to similar techniques.

In Graves *et al.*'s empirical study, safe and dataflow had similar fault detection coverage, but dataflow was less cost-effective due to the need of additional analysis [2].

There are a number of different subsets of selection techniques, many of them are identified in the study by Yoo and Harman [4]. One of them is the modification-based technique, where tests are selected by source code analysis, identifying the changes done to a system and targeting affected units.

The prioritization technique will arrange tests in the test suite for early fault detection [4]. The purpose of this is not necessarily to reduce the test suite, but to prioritize the test cases to increase the possibility of early fault-detection. A possible drawback of this method is that the prioritization requires input from a user, making it biased [4].

B. Software Context Matters

A case study was done on regression test technology adoption [12], where it is suggested that the RTST should be chosen with the context of the SUT in mind. The authors argue that a systematic approach should be taken when introducing a new RTST. The approach stems from the concept of evidence-based software engineering, and by providing practical examples, Engström *et al.* makes this concept more accessible for a wider audience. The purpose of Engström *et al.*'s study was to give advice regarding a systematic approach to evidence-based regression testing. Engström *et al.* discusses the aspects of increased automation in RTS, but also provides a foundation for choosing a RTST.

C. Continuous Integration

In order to uphold quality in a cost-effective manner, reliance on CI becomes more important for a growing system [8]. Applying RTS to CI with test suite data sets have seen improved cost-effectiveness [7], [8]. Many solutions are tailored prioritization techniques applied to improve cost-effective CI, to provide early fault detection, by informing developers quicker for introduced software faults.

D. Large Scale Software Systems

The system chosen for this study is a LS3. These types of systems are complex to maintain and evolve, because of the many dependencies larger systems tend to have [11]. We have the type of system in mind when conducting this study, as we have learned that the context of the SUT is important when selecting a RTST. Orso *et al.*'s experimental study provides an example of a tool, specifically scaled for LS3 [5], and implements a safe RTST.

Orso *et al.*'s RTST consists of: *Partitioning*–the tool identifies sections of the code that needs further analysis. *Selection*–an in-depth analysis is done in order to select appropriate test cases to run.

RTSTs all have distinct trade-offs, often between saving time or accuracy of the selection. The aim of creating a subset of test suites is the prospect of being more efficient. Or as Orso *et al.* states it [5]:

> In general, for an RTS technique to be cost-effective, the cost of performing the selection plus the cost of rerunning the selected subset of test cases must be less than the overall cost of rerunning the complete test suite

Re-running the entire test suite comes with high costs for a LS3, but comes with encompassing test coverage and high reliability [2]. Arguably it is not the most efficient way of conducting RT, as it does not leave out tests that have not been affected by the new modification to the system.

There is plenty of prior research which examines and aims to find the most effective RTST [1], [2], [5]. Similarly, most research on this topic has performed RTS on smaller software systems.

RT studies in the 1990's reveal advantages using small components to isolate the test suites into segments to reduce the testing scope for the overall suite, and lower the cost of RT [9].

The behavior is retrieved from domain experts working with the LS3. Afterwards, the most appropriate RTST will be evaluated and compared.

The current implementation of the RTS is in the form of an algorithm. The algorithm generates a Dependency List (DL) based on the information from commits to the SUT. The DL contains files on what sections of the SUT has been modified, and needs to be tested.

This type of RTST has room for improvements in time-efficiency and reduction of false positives in the DL. In this study, the term false positive refers to dependencies that are included in the DL, but should not be, as it does not contribute to fault detection. The existence of false positives is an issue, as it would require more resources than necessary to perform RT.

This will be investigated using different RTSTs, comparing their efficiency and precision through metrics applied on a LS3. To compare the RTST, we need to create a second RTST, resulting in an artefact.

Introducing a commit will prompt the system to generate a list of affected SCs. By using build files which are located in each SC, they can specify their build dependencies [8]. The build files are recursively used to produce a DL. SCs can use this build feature, those that do not, are always included in the DL.

Parallel RTS is an important factor in achieving high speeds for a LS3. Also, prioritization techniques become less fruitful the more things you are able to execute in parallel.

III. Research Methodology

The first stage of this study is to conduct an investigation of an RTST. This artefact is a DL generation algorithm, referred to as Algorithm–Existing (A_E). The purpose of this investigation is to further our understanding of this implementation and to classify it.

The following step is to develop three algorithms for RTS: Algorithm–Minimization (A_M), Algorithm–Filter (A_F), and Algorithm–Random (A_R). The new artefact will consist of the application of A_F, A_E and A_M in sequence, depicted in Fig. 3, and refereed to as Algorithm–Artefact (A_A). A_A will be developed for a LS3.

Lastly, we will conduct an empirical study to compare the results of A_E, A_R, and A_A. As well as make a statement as to which solution is best suited for the LS3, taking cost effectiveness and RT coverage into account. See table I for RT coverage metrics.

TABLE I
RT COVERAGE METRICS

Metric	Description
True positives	Number of files in the DL that indeed needs to be analyzed
False positives	Number of files that do not need dependency analysis
False negatives	Number of files missing in the DL

Additionally, *Results* also include the metric of time to generate the DL.

Due to the fact that we will design an artefact in iterations in this study, we decided to align our process with a design science research approach [13]. See Fig. 1 for our process.

The new artefact, A_A, should create a DL, that includes all files that needs to be tested, and reduce the inclusion of false positives.

We keep the commit sample constant when testing the algorithms, in order to ensure comparable results.

These assumptions led to the formulation of the following null hypotheses, which we are aiming to reject. Each hypothesis is formulated in a way so that the results from the data analysis will either allow us to reject them, or fail to reject them. The alternative to each hypothesis is that there instead is no difference between the result, and that they are equal.

$\boldsymbol{H_0}$ **1** $\mathrm{Time}(A_A) > \mathrm{Time}(A_E)$

Using A_A to generate the DL, will take more time than using A_E.

$\boldsymbol{H_0}$ **2** $\mathrm{FalsePositive}(L_1) < \mathrm{FalsePositive}(L_2)$

False positives in the DL generated by A_E, will be fewer than those in the DL generated by A_A.

$\boldsymbol{H_0}$ **3** $\mathrm{RTCoverage}(L_1) > \mathrm{RTCoverage}(L_2)$

The A_E technique is better at including more files in the DL that are supposed to be included (true positives).

By performing interviews with the stakeholders, we will investigate the current RTST.

Peffers *et al.* [13] state a method of obtaining information through interviews:

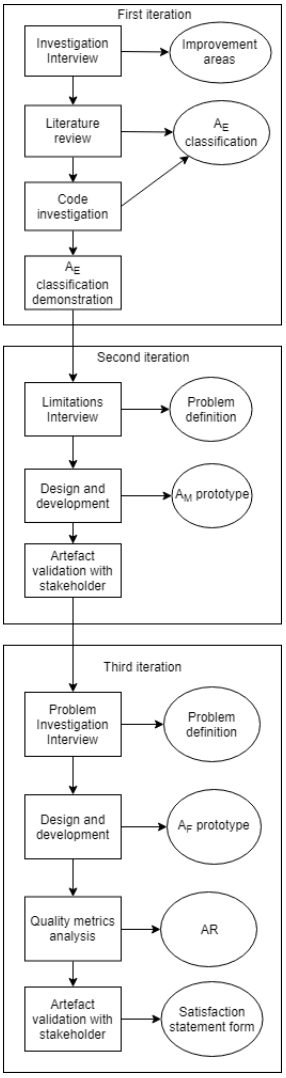

Fig. 1. DSRM process, activities and artefacts.

> "Details about the company's development environment [...] specifics about the component design, and the classification of certain code stubs, were obtained through structured interviews." [13]

Structured interviews can provide details about component design and classifications of code [13]. While interviews that are exploratory and semi-structured assist with understanding the context of the system [12].

The structured context checklist was proposed by Petersen and Wohlin [10], and used by Engström *et al.* [12]. The checklist can assist with learning about the context, purpose, and constraints [12]. A subset of the checklist is adopted for our study, and we aim to learn about the following: How long the system has been released, driving quality aspects, and measurement of system size or complexity.

In combination to this initial interview, a *code investigation* is performed on the RTS implementation, A_E, that generates the DL. The purpose of the investigation is to further our understanding, and to classify it under any of the RTSTs described in section II. Additionally, we will be able to answer *RQ 1*.

A *classification demonstration* is performed with a domain expert. The purpose of the demonstration serves to provide the classification to them, and identify if our understanding of A_E is accurate.

The RTST used when implementing A_A will be selected based on it's strengths against the limitations we will find during the *code investigation* phase. A solution fulfills the objective in a sufficient manner, and should be based on prior knowledge, derived from related literature and the problem statement [13]. Because the design of A_A will be done in iterations, the basis for the design choices are derived from A_E, previous literature, and feedback on design plans for A_A from stakeholders during an initial evaluation interview.

As stated previously, the artefact is developed in iterations. The second iteration results in the A_M prototype, and the third iterations results in the A_F prototype.

The three algorithms, A_E, A_M and A_F are combined to create the artefact A_A, see 3.

Additionally, a framework for our study is developed alongside A_A and A_R. The framework records and stores measurements from the algorithms it is run on.

The artefact, A_A, is evaluated by examining the resulting quality metrics. These metrics are collected by testing A_E, A_A and A_R on a sample of commits and the SUT. By performing these tests, it allows us to discover aspects of A_A that could be improved.

The artefact A_A is evaluated by performing *an artifact validation* interview with stakeholders. They are asked to fill out a form with Likert scale questions.

We present the stakeholders with the artefact A_A, it's design and evaluation, and the comparison between A_E and A_A.

A. Data Collection

TABLE II
DATA POINTS

Data point	Description
Dependencies	Number of files in the DL
Total files	Number of files that all DL's SC depends on
Generation time	Time in seconds to perform dependency analysis
Committed files	Number of files we perform dependency analysis on

Table II outlines the data we collect, and how we describe the data points in the paper.

The data is normalized by ignoring dependencies which are not affected by the commit, and by any RTST implementation. We perform this normalization in order to interpret that zero dependencies, would mean that an individual commit would produce no dependencies.

Commits can modify a large variety of file types. Many that can be difficult to perform deeper analysis on, given that, we choose C/C++ source code. And select a set of recent commits that modify that type of file. Thus resulting in our population.

The sample size is based on two assumptions: The time to generate the DL will not vary depending on the commit. And,

the RT coverage for a commit will be consistent, given the same algorithm.

The generation time to run the algorithms limits the sample size possible. Using a confidence level of 90% for our one-tailed t-test, we interview the stakeholder and make an estimation for the sample size. We decide on a sample size of 500 commits to reflect the population. In order to reflect the current affects of the RTST, the commits are recently sampled from the population. The population size is undetermined, because it is ever growing. The sample will be kept consistent for A_E, A_R and A_A.

We choose to divide the population of commits into one homogeneous subgroup based on coding language, thus narrowing our scope to include only the sub-group of commits that include C/C++ code. From this subgroup, the sampling was then randomly conducted.

B. Data Analysis

The initial plan for the data analysis was to compare the results for each RTST implementation to a baseline. However, this was deemed not possible, as we found that we can not determine what is a true positive. This is because we do not mean to actually run the test suites that the DL generate.

The resulting analysis will provide the following metrics for each RTST: dependencies, total files, generation time, and committed files.

We perform statistical analysis using the data from running A_E, A_A and A_R. The results from the analysis are used to do hypothesis testing, to reject or fail to reject the hypotheses. For this, we compare numerical values between two sets of data, and we want to determine if each data point from A_A is statistically different from A_E, and we only want to know if A_A is better, so we choose a one-tailed t-test.

We also use descriptive statistics to outline a summary of the entire data set, comparing the benefits or drawbacks of each technique.

IV. Results

A. A_E Limitations Interview

By performing this interview, we aim to understand it's applied software context, as well as discover improvement areas for A_E, and use the findings to construct the A_A artefact.

1) Identified Limitations: The interview yields three improvement suggestions to mitigate limitations of A_E: It takes a significant amount of time to generate the DL, but it will in most cases, not miss dependencies. It is proven that A_E can miss tests that would reveal software faults, but the test coverage is satisfactory as it does not occur often, while it can not guarantee to be safe. The outcome from sub par test coverage is expensive, so it is better to test more than to test less.

For trivial changes, i.e., code comments, A_E would produce a component level test suite with all its dependencies. As A_E does not look at the change set, and only the files which were changed.

B. A_E Investigation

During the investigation, we inspected the code implementation together with the domain expert.

Algorithm 1: A_E

```
foreach SC in SUT do
    generate dependencies;
    foreach changed file do
        if changed file in dependencies then
            include SC in DL;
```

Fig. 2. A_E's implementation

Based on the interview, we conclude that the analysis is somewhat static: if a non-functional statement, like a comment in the code is modified, the DL will include all dependencies to that file and run tests on them.

Performing early fault detection after a selection technique is common for test case prioritization [4]. Additionally, test suite prioritization has been performed on a similar case to ours [7]. But it was confirmed by the domain experts as being a less fruitful approach for the applied context, where they use large amounts of parallelization.

The time complexity to generate the DL for A_E was found to be mostly static on different commits. The largest effect on time for A_E is the number of SCs it needs to identify dependencies for.

Even though the algorithmic complexity is $O(SC * changedfile)$, checking each changed file is significantly faster than checking each SC.

A_E generates a DL for all SC, in our tests, when performed linearly, the bottlenecks become more prominent as A_E is designed to parallel each DL generation for every SC.

A_E is designed to handle a large set of programming languages and configurations. These are important factors, but can also increase the time of generation as all the cases needs to be checked.

C. A_E Classification

The results from the investigation leads us to conclude that the current implementation, A_E, is a selection technique, more specifically, a modification-based technique.

D. Designing A_A

Unlike A_E, that conducts one single activity to perform RTS, the new approach, A_A, composites three techniques that are split into different stages to reduce the overall test time. See Fig. 3.

The first step is to re-use A_E, to perform partitioning on the test suite, and select a broad number of files, and generate a DL that includes false positives.

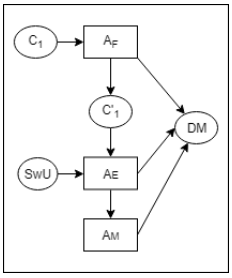

Fig. 3. Diagram of A_A

1) Minimization: Secondly, we perform a fine-grain minimization over the partitioned results, removing false positives from the DL using the commit change set. This algorithm is called A_M.

We extend A_E by applying a method on the commit which removes changes that do not impact the resulting compiled code, such as white spaces, comments and print statements [14]. The extension will minimize the existing DL, but increase the generation time of performing the RTS. The goal being to reducing the overall RT time.

A_M is implemented in Python, and described using pseudo code in Fig. 4. A motivation example then follows, outlining a case where the changes would be ignored (see Fig. 5).

Algorithm 2: Minimization, ignores cosmetic changes

```
foreach line do
    Normalize line;
    if modified comment or logging then
        continue;
    if start of multiline comment then
        in multiline comment block;
        continue;
    if in multiline comment then
        if end multiline comment then
            out of multiline comment block;
        continue;
    include change;
```

Fig. 4. A_M's implementation

A_M will exclude tests and dependencies for commits that only include changes to comments and white space. The algorithm will also omit cosmetic changes.

2) Filter: A filter, A_F, is implemented in the beginning of the test flow, to filter out dependencies that are in the test directory.

A secondary approach, A_F, is developed using a cost-reducing filter. The filter approach is performed before A_Es dependency analysis, with the aim to reduce the time and cost of RTS. If the committed files modify test code, the files do not need to be analyzed by A_E, and only include the SC in the DL. The filter is implemented in bash, and described using pseudo code in Fig. 6.

```
+ /*
+ * Comment
+ */
  if (a) {
+     TRACE(0, "...");
-     return 1;
+     return 1; // Comment
  } else {
      return 2;
+     // Comment
  }
```

Fig. 5. Minimal example of a file which would be minimized, and not including in the DL using A_M. Additions are marked with a plus sign, and deletions with a minus sign.

Algorithm 3: A_F, Differentiate changes in test files, to other changes.

```
foreach changed file do
    if not in test directory then
        continue;
    if not find build file for the changed file then
        continue;
    Remove file from C1;
    Add build file as a dependency;
```

Fig. 6. A_F's implementation

The goal of this filter is to reduce the number of files we need to look up in the dependency analysis, further, reducing the time it would take to generate. And it if filters out all the files, no dependency analysis is performed. Fig. 7. is a motivation example over the inputs and outputs of A_F.

3) Random: In order to examine the potential cost efficiency of performing dependency analysis in A_E, as well as the new implementation A_A, a random solution was developed, A_R. Fig. 8 provides the pseudo code for the bash implementation.

The random technique, A_R, will select a random dependency, a random amount of times, and add to a DL. Finally, A_R will remove duplicate dependencies from the DL.

E. Initial Evaluation

An initial evaluation is performed with the domain expert. We explain our findings for A_E so far, as well as the basis for

```
AF input files:
./SC/test/file1.cc
./SC/test/file2.cc
./SC/test/file3.h
./SC/src/file4.cc

AF output:
   Files: ./SC/src/file4.cc
   DL:    ./SC
```

Fig. 7. Overview A_F's filtering approach, which replaces the test files with the build file and does not perform a dependency analysis on.

Algorithm 4: A_R

foreach *random number of times* **do**
    include random dependency in dependency list;
remove duplicate dependencies;

Fig. 8. A_R's implementation

the design choices for A_M.

One suggestion was an initial check, to verify if the commit required testing at all. Along with a check to verify if the change set would be valid in the affected programming language. This has the limitation of only being effective if the check was quicker than A_E, and if it reduced the the total time spent performing RT.

We also discussed the potential use of early fault detection. However, this idea was scrapped, as the importance of the overall RTS time is more valuable and measurable.

During the evaluation, we also found additional limitations: A_E needs to be maintained to include new file extensions/ types every time new file types are occur in the dependencies. If the algorithm is not edited, files of these types will be not included in the DL.

They approved the idea of smart selection, which would be the extension of A_E.

F. Prototype Evaluation

An evaluation of the A_M prototype is performed with the domain expert. They validate the behavior of A_M. The domain experts provide additional ideas for the next iteration for improving the A_M algorithm.

In the SUT there are commits that only make changes in test code, and does not require a dependency analysis, and should only test that SC. Since testing only the affected file is enough to validate the behavior.

G. Data Collection

The sample size is decided to be 500 commits. We created a subgroup from the population, in order to get a sample that is mainly C/C++ code, but still random.

The sample–data is collected for all three techniques: A_E, A_A and A_R.

For the sum of the entire sample–data, A_A produced 14.7% less dependencies. A_A also reduced the committed files by 151, and we determine these to be false positives. In the sample–data, the number of commits with no dependencies, rose from 57 in A_E to 65 in A_A.

A_A reduced the dependencies in 80 commits. A_A also increased the dependencies in 18 commits. In most cases, with a single additional dependency.

If we only look at the commits that differed between A_E and A_A, the mean difference was one dependency.

H. Data Analysis

Here we describe the results from performing statistical analysis, and descriptive statistics on the sample–data.

The collected data is checked for normality, and does show a normal distribution, see Table III.

TABLE III
TEST OF NORMALITY (SHAPIRO-WILK)

			W	p
A_E dependencies	-	A_A dependencies	0.389	$< .001$
A_E committed files	-	A_A committed files	0.444	$< .001$
A_E generation time	-	A_A generation time	0.433	$< .001$
A_E total files	-	A_A total files	0.408	$< .001$

Afterwards we perform a statistical hypothesis test, comparing the collected data described in Table II.

The results of the t-test are shown in Table IV. The t-test indicate that there was no significant difference between A_A and A_E. Thus we fail to reject all three null hypotheses.

We do not perform a t-test on A_R because it is only used as a benchmark.

TABLE IV
PAIRED SAMPLES T-TEST

			t	p
A_E dependencies	-	A_A dependencies	6.009	$< .001$
A_E committed files	-	A_A committed files	10.976	$< .001$
A_E generation time	-	A_A generation time	5.323	$< .001$
A_E total files	-	A_A total files	6.372	$< .001$

Table V describes the descriptive statistics, including mean and standard deviation for all three techniques. We analyzed how A_R compared to A_E and A_A. A_R's dependencies had a median of 71.424, while A_E and A_A had a median of 3.776 and 3.292, respectively. The amount of dependencies are significantly higher when a DL is generated by A_R, but the generation time had a median of 100.600 seconds, which is less then that of A_A's at 195.165 seconds and A_E's at 179.912 seconds.

Table V shows that the standard deviation is lower for A_A for every data point, meaning that A_A is closer to the average, and less spread out compared to A_E.

TABLE V
DESCRIPTIVE STATISTICS

	Mean	SD
A_E dependencies	3.776	7.220
A_A dependencies	3.292	7.118
A_R dependencies	71.424	34.823
A_E committed files	8.924	21.787
A_A committed files	6.560	17.685
A_R committed files	8.922	21.788
A_E generation time	195.164	75.173
A_A generation time	179.912	75.316
A_R generation time	100.600	60.690
A_E total files	1032.460	2037.672
A_A total files	827.524	1952.751
A_R total files	100022.262	5087.445

Looking at Fig. 9, we can see that the number of dependencies varies slightly between A_E and A_A. The histogram also shows that A_A produced less dependencies more frequently.

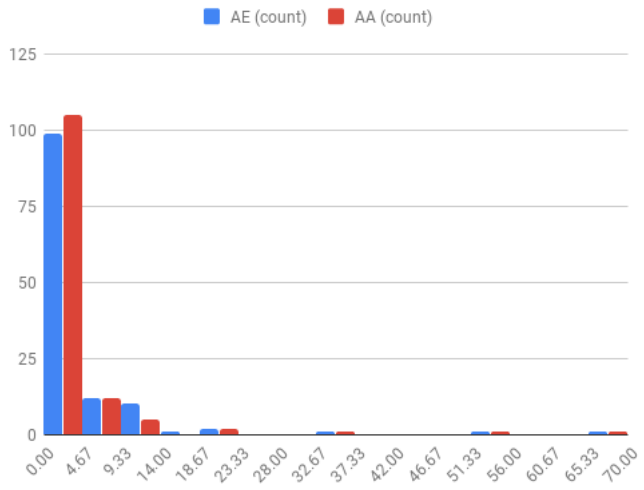

Fig. 9. Histogram over A_E and A_A's Dependencies

I. Validation With Stakeholder

An interview is held with the stakeholders, in order to validate the behavior of the artefact and present the results. We also tell them about the behavior of A_A, and show them the results from the data analysis. Based on this knowledge, the stakeholder is able to answer the survey, see Table VI.

TABLE VI
STAKEHOLDER SURVEY AND ANSWERS

The subset of commits reflect the norm.	Agree
The new solution will be used in the future.	Neutral
The new solution meets the company's needs.	Neutral
The new solution is maintainable/easy to maintain.	Agree
I am happy with these results.	Neutral

The satisfaction level was mostly positive, but mixed. The stakeholder expressed that they would have been more happy with an artefact that was ready to be implemented for the SUT.

The stakeholders express that they might be interested in using parts of A_A, A_F, since A_F is easier to maintain, implement and extend.

What constitutes as a cosmetic change, depends on the programming language. The implementation of A_M is therefor programming language dependent, and thus is less generalizable for a larger array of programming languages. In most cases, to be efficient with the implementation, it requires domain knowledge of the source code.

A_F is designed to match the specific directory structure of the SUT, thus making A_F easy to generalize, and apply on to other SUTs. It also does not have the same limitations as A_M, and does not depend on the programming language.

V. Discussion

In this study we connect RTS research with dependency analysis research. This connection lead to complications in applying RTSTs similarly to that of literature.

We acknowledge that no RTST solves every use case.

The developed approaches are lightweight, and shows improvements for an existing RTST. The investigation, and subsequent interviews provided valuable insight not found in documentation, we highlight this as a critical step in the resulting artefact and study.

A. Related Work

The design choices for A_A were inspired by related literature, as well as limited by the existing solution, A_E. We deemed it more resource-effective to re-use the A_E solution. We decided to partially follow Orso *et al.*'s *Partitioning and Selection* technique, as it is scaled for an LS3. This technique combines two RTST in order to get the benefits from both [5]. As A_E is a modification-based selection technique, and already has the benefits and drawbacks from that type of RTST, we decided that the second step would be a minimization technique, removing from the DL, those dependencies that were selected and included by A_E [4].

We partially extend Vokolos *et al.*'s conclusion with lightweight minimization [14]. They highlight efficiency for their RTST, which uses canonical form, and claim that it does not increase time for RT. Similarly, these observations can be made with A_A.

The prioritization technique described by Yoo and Harman was also considered. However, after deliberating with the stakeholders, we decided not to try out such a technique. Upon describing the benefits and drawbacks of such a technique, the stakeholders deemed it not beneficial for their context [4]. We also agree that the risk of adding selection bias to their test routines, would pose a bigger drawback, than any possible benefits would add.

B. RQ1

To answer *RQ 1* we conducted three interviews, and a code analysis. The following are improvement areas we did not improve upon using A_A.

Presented by the stakeholders were other optimization, or improvements to A_E. These improvement were highly dependent on the SUT. We declined some suggestions, which relate to the domain of the SUT, and/or programming language applied, due to the specificity and small generalize-ability. The resulting artefact adopts two suggestions from the stakeholders to design A_M and A_F.

We extended on to A_E to create A_A. Doing so, we can only reduce the existing DL using the two approaches (A_M, and A_F). We argue the limiting factor for the number of false negatives in the findings, do not change, due to still using A_E. No improvements have been made to expand the DL, thus false negatives have not been reduced.

C. RQ2

With *RQ 2*, we set out to investigate the differences between two RTSTs. In order to answer *RQ 2*, we intended to look at a few metrics.

We set a few guidelines as to what qualifies as a false positive (see Section III). According to these, A_E included 151 false positives.

With the collected data, there is no way of checking if the results from A_A included any false positives. Because of this, we assume that A_A can be improved.

Investigating false negatives was not feasible during this study, therefor we acknowledge this as a limitation in our study.

Similarly, true positives are also difficult to analyze, as we do not have the means to analyze to see if the resulting DL completely covers all necessary dependencies.

The hypothesis testing showed no significant results, and we were not able to reject any of the three null hypotheses.

Additionally, we compared the standard deviation for all RTSTs in Table V. We can not say whether a higher or lower standard deviation is better for generation time, as A_A could produce a very fast dependency analysis given a commit of only test changes appropriate for A_F, thus giving us a larger standard deviation.

We also implemented a random technique, A_R. The purpose of this was to have a worst-case benchmark, to compare against our findings. This implementation had a lower generation time. However, this was at the cost of test coverage accuracy, according to the metrics we looked at. See Fig. 8 for the pseudo code algorithm.

The three different RTST have both drawbacks as well as benefits. The effectiveness of a RTST is dependant on the system context, in which it is implemented on [12]. This resonates with statements made in related work in the field.

D. Validity Threats

1) Classification of A_E: The investigation relies on the interpretation of A_E. A_E can consist of several techniques, and possibly not fall under a single definition as defined in literature.

2) Variations: Software in general is prone to intermittent variations, we see this in the variations on the generation time for the DL when conducing the study. As we can not reliably ensure that nothing will vary the results on the same hardware, variations are mitigated by running the tests on the least active development hours.

3) Generalizability: The result from the data analysis can only be applied for the context of this study, we only draw conclusions and make claims about the specific SUT. However, the RTST and the investigation thereof, can be generalized and be applied globally.

4) Analytic Bias: There is also risk for bias in the questionnaire with the stakeholders. As the questions followed a Likert scale format, there is a risk for a central tendency bias, where the stakeholder might avoid extreme answers, as well as try to align their answers with what they think the researchers want to hear [15].

5) Sample Bias: In the data collected for A_E, 57 commits were found to produce no DL. Two theories for this would be that some commits did not modify any dependent files, or if false negatives exist in A_E, which we did look for.

In addition, the commits sample might not reflect the population, as commits can vary along several dimensions, i.e., if randomly sampled from a population, it might give results that provides little benefit from an analysis point of view. In this case, we tried to mitigate this issue by focusing only on the part of the population that has C/C++ code.

VI. Conclusion

Regression Testing (RT) becomes more important with Large Scale Software Systems (LS3). The larger the test suite is, selecting what to test becomes more important, as testing everything is no longer cost-effective. In literature we can see that many authors have proposed Regression Testing Selection Techniques (RTST).

In this study, we explored an existing RTST, Algorithm–Existing (A_E), and classified it as a modification-based selection technique. We extended the existing Regression Testing Selection (RTS) with two additional approaches: Removal of cosmetic changes introduced in the commit by minimizing the selection, and filtering out files based on file names and separate changes in test files from other changes.

Our statistical analysis, however, did not find any significant results. Nevertheless, the analysis points towards a total 14.7% reduction in the sum of all Dependency List (DL), when comparing A_E with Algorithm–Artefact (A_A).

The interviewed domain experts described limitations with the current implementation. They outlined maintainability as important, and describe cases that leads to missed dependencies.

A new solution, A_A, was developed, and tested on a LS3, minimizing commits that only included changes to comments, and filtering out dependencies that are tests. With this approach, we were able to somewhat reduce the DL, although not finding any statistically significant improvements. However, we did see a reduction in the standard deviation between the two implementations, which could be beneficial if implementing this in current systems.

In this study, we have presented potential benefits - as well as limitations - of two RTST, as well as a random technique. Additionally, we added to the existing research on applying RTST to LS3.

We want to highlight potential future work on connecting RTST research with dependency analysis research. According to us, this is an area that would benefit from further investigation. One possible area would be on how to apply more fine grained analysis on dependency analysis.

Glossary

A_A Algorithm–Artefact. 3–9
A_E Algorithm–Existing. 3–9
A_F Algorithm–Filter. 3, 4, 6, 8, 9
A_M Algorithm–Minimization. 3, 4, 6–8
A_R Algorithm–Random. 3–7, 9

CI Continuous Integration. 1, 2

DL Dependency List. 3–9
DSRM Design Science Research Methodology. 1, 4

LS3 Large Scale Software Systems. 1–3, 8, 9

RT Regression Testing. 1, 3, 5–9
RTS Regression Testing Selection. 1–9
RTST Regression Testing Selection Techniques. 1–5, 8, 9

SC Software Component. 2–7
SUT System Under Test. 2–5, 7–9

REFERENCES

[1] J. A. Mayan and T. Ravi, "Structural software testing: Hybrid algorithm for optimal test sequence selection during regression testing," *International Journal of Engineering and Technology (IJET)*, vol. 7, no. 1, 2015.

[2] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[3] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for c++ software," *Software Testing, Verification and Reliability*, vol. 10, no. 2, pp. 77–109, 2000.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[5] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 29, 2004, pp. 241–251.

[6] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[7] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 235–245.

[8] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, IEEE Press, 2017, pp. 233–242.

[9] D. Binkley, "Reducing the cost of regression testing by semantics guided test case selection," in *Proceedings of International Conference on Software Maintenance*, IEEE, 1995, pp. 251–260.

[10] K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE, 2009, pp. 401–404.

[11] M. Skoglund and P. Runeson, "A case study of the class firewall regression test selection technique on a large scale distributed software system," in *2005 International Symposium on Empirical Software Engineering, 2005.*, IEEE, 2005, 10–pp.

[12] E. Engström, R. Feldt, and R. Torkar, "Indirect effects in evidential assessment: A case study on regression test technology adoption," in *Proceedings of the 2nd international workshop on Evidential assessment of software technologies*, ACM, 2012, pp. 15–20.

[13] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[14] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *Reliability, quality and safety of software-intensive systems*, Springer, 1997, pp. 3–21.

[15] D. Bertram, "Likert scales," *Retrieved November*, vol. 2, p. 2013, 2007.